# Error-Detection Codes: Algorithms and Fast Implementation

Gam D. Nguyen

**Abstract**—Binary CRCs are very effective for error detection, but their software implementation is not very efficient. Thus, many binary non-CRC codes (which are not as strong as CRCs, but can be more efficiently implemented in software) are proposed as alternatives to CRCs. The non-CRC codes include WSC, CXOR, one's-complement checksum, Fletcher checksum, and block-parity code. In this paper, we present a general algorithm for constructing a family of binary error-detection codes. This family is large because it contains all these non-CRC codes, CRCs, perfect codes, as well as other linear and nonlinear codes. In addition to unifying these apparently disparate codes, our algorithm also generates some non-CRC codes that have minimum distance 4 (like CRCs) and efficient software implementation.

**Index Terms**—Fast error-detection code, Hamming code, CRC, checksum.

## 1 Introduction

Efficient implementation of reliable error-protection algorithms plays a vital role in digital communication and storage because channel noise and system malfunction introduce errors in received and retrieved messages. Here, we focus on binary error-detection codes that have low overhead and minimum distance $d \leq 4$. Popular error-detection codes used in practice include CRCs that are generated by binary polynomials such as $X^{16}+X^{15}+X^2+1$ (called CRC-16) and $X^{16}+X^{12}+X^5+1$ (called CRC-CCITT).

An $h$-bit CRC generated by $G(X)=(X+1)P(X)$, where $P(X)$ is a primitive polynomial of degree $h-1$, has the following desirable properties [1]. The CRC has maximum codeword length of $2^{h-1}-1$ bits and minimum distance $d=4$, i.e., all double and odd errors are detected. This code also detects any error burst of length $h$ bits or less, i.e., its burst-error-detecting capability is $b=h$. The guaranteed error-detection capability of this $h$-bit CRC is nearly optimal because its maximum codeword length almost meets the upper bound $2^{h-1}$ and its burst-error-detecting capability meets the upper bound $h$. The CRC is also efficiently implemented by special-purpose shift-register hardware.

Although CRCs have nearly optimal properties and efficient hardware implementation, many binary non-CRC codes are proposed as alternatives to CRCs. These codes, developed over many years and often considered as unrelated to each other, do not have the CRC's desirable properties. Such non-CRC codes include weighted sum code (WSC), Fletcher checksum (used in ISO), one's-complement checksum (used in Internet), circular-shift and exclusive-OR checksum (CXOR), and block-parity code (Fig. 1). See [4], [5], [9], [14] for implementation and performance comparisons of CRCs and these non-CRC codes. Perhaps the key reason for the appearance of the non-CRC codes is that CRCs are not very efficiently implemented in software. Software complexity refers to the number of programming operations and hardware complexity refers to the number of gates required for code implementation. Investigations reported in [4], [9] indicate that software processing of CRCs is slower than that of the non-CRC codes. Thus, it is desirable to design error-detection codes that are reliable and of low complexity. One code is better than another if, for a fixed number of check bits $h$, it has larger minimum distance $d$, larger burst-error-detecting capability $b$, longer maximum codeword length $l_{max}$, and lower complexity.

An important performance measure of a code, which is not addressed in this paper, is its probability of undetected error. For the binary symmetric channel, this probability can be expressed in terms of the weight distribution of the code. In general, the problem of computing the probability of undetected error is NP-hard [7]. Some methods for calculating or estimating this probability are given in [7].

Because the minimum distance $d$ is often considered the most important parameter, Fig. 1 ranks CRC as the best code, WSC the second best, and so on. Although the WSC, Fletcher checksum, and CXOR are defined only for an even number of check bits $h$, both even and odd $h$ can be used for the other codes. The CRC, WSC, and Fletcher checksum can be extended to have infinite length, but their minimum distances all reduce to 2. Some discussions of burst-error-detecting capability $b$ are given in Appendix C (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm). In this paper, we focus on code implementation by means of software. Because computers can process information in blocks of bits (e.g., bytes or words), codes having efficient software implementation should also be processed in blocks of bits. Then, it is natural to express code lengths in terms of the number of blocks $n$ and each block is $s$ bits, i.e., the total number of bits is $ns$. Most modern processors can efficiently handle block

• *The author is with the Information Technology Division, Naval Research Laboratory, Washington, DC 20375. E-mail: nguyen@itd.nrl.navy.mil.*

*Manuscript received 27 Mar. 2003; revised 26 Feb. 2004; accepted 30 July 2004; published online 16 Nov. 2004.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 118495.*

| Report Documentation Page | *Form Approved* *OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| 1. REPORT DATE **26 FEB 2004** | 2. REPORT TYPE | 3. DATES COVERED **00-00-2004 to 00-00-2004** |
| 4. TITLE AND SUBTITLE **Error-Detection Codes: Algorithms and Fast Implementation** | | 5a. CONTRACT NUMBER |
| | | 5b. GRANT NUMBER |
| | | 5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER |
| | | 5e. TASK NUMBER |
| | | 5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Information Technology Division,4555 Overlook Avenue SW,Washington,DC,20375** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR'S ACRONYM(S) |
| | | 11. SPONSOR/MONITOR'S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT **see report** | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT **unclassified** | b. ABSTRACT **unclassified** | c. THIS PAGE **unclassified** | **Same as Report (SAR)** | **11** | |

| Error-detection codes | $d$ | $b$ | $l_{max}$ |
|---|---|---|---|
| CRC | 4 | $h$ | $2^{h-1}-1$ |
| WSC | 4 | $h/2+1$ | $h2^{h/2-1}$ |
| Fletcher checksum | 3 | $h/2-1$ | $h2^{h/2-1}+h$ |
| Block-parity code | 2 | $h$ | infinity |
| 1s complement checksum | 2 | $h-1$ | infinity |
| CXOR | 2 | $h/2+1$ | infinity |

Fig. 1. Error-detection capabilities of binary codes ($d =$ minimum distance, $b =$ burst-error-detecting capability, $h =$ number of check bits, $l_{max} =$ maximum codeword length).

size $s = 8, 16, 32, 64$ bits. General-purpose computers and compilers are increasingly faster and better. Thus, software algorithms become more relevant and desirable. Software algorithms are increasingly used in operations, modeling, simulations, and performance analysis of systems and networks. An important advantage of software implementation is its flexibility: It is much simpler to modify a software program than to modify a chip full of hardwired gates and buses.

In this paper, we present a general algorithm and its systematic versions for constructing a large family of binary error-detection codes (Section 2). This family contains all the codes in Fig. 1 and other linear and nonlinear codes for error detection. We unify the treatment of these seemingly unrelated codes by showing that CRCs and the non-CRC codes all come from a single algorithm (Section 3). Further, the algorithm can produce some non-CRC codes that are not only reliable (i.e., having minimum distance 4 as CRCs), but also have fast software implementation (Section 4). We then summarize and conclude the paper (Section 5). The paper is supplemented with appendices (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm) that include theorem proofs, code segments implemented in C programming language, as well as discussions of CRCs, WSCs, and CXORs. The preliminary version of this paper is presented in [10].

### 1.1 Notations and Conventions

We consider polynomials over only the *binary* field GF(2), i.e., the polynomial operations are performed in polynomial arithmetic *modulo* 2. Let $A = A(X)$ and $B = B(X)$ be two polynomials, then $A \bmod B$ is the remainder polynomial that is obtained when $A$ is divided by $B$ with $\deg(A \bmod B) < \deg(B)$. To ease the presentation of many different codes (which can result in a large number of parameters), we adopt the following sweeping conventions. A *j*-tuple $(a_0, \ldots, a_{j-2}, a_{j-1})$ denotes the binary polynomial $a_0X^{j-1} + \ldots + a_{j-2}X + a_{j-1}$ of degree less than *j*. In this paper, lowercase letters (such as $h$ and $a_0$) denote nonnegative integers. The letters $C$ and $C_1$ denote codes, other uppercase letters (such as $A$ and $Q_i$) denote polynomials (or tuples), and $X$ denotes the variable (or indeterminate) of these polynomials. Further, the variable $X$ will be omitted from all polynomials, i.e., $A(X)$ will be denoted as $A$. We denote $u^i$ as the $i$-tuple whose components are all $u$s, $u \in \{0, 1\}$. The notation $(l, k, d)$ denotes a systematic code with $l =$ code length, $k =$ information block length, and $d =$ minimum distance. Finally, if $Y_1$ and $Y_2$ are $m_1$-tuple and $m_2$-tuple, respectively, then $Y = (Y_1, Y_2)$ denotes the concatenation of $Y_2$ to $Y_1$, i.e., $Y$ is an $(m_1 + m_2)$-tuple. Note that $Y$ can also be written as $Y = Y_1X^{m_2} + Y_2$. For ease of cross-referencing, we usually label blocks of text as "Remarks." These remarks are integral parts of our presentation and they should not be viewed as isolated observations or comments.

## 2 A General Algorithm for Error-Detection Codes

In this section, we define a binary code so that each of its codewords consists of $n$ tuples $Q_0, Q_1, \ldots, Q_{n-1}$, each tuple is $s$ bits. This code is not necessarily systematic and is formulated abstractly to facilitate the development of its mathematical properties. For practical use, we then construct systematic versions of the code. Fast software versions of the code will be presented later in Section 4. It is important to note that $Q_i$ is an uppercase letter, so, by our conventions, $Q_i$ is a polynomial of the variable $X$, i.e., $Q_i = Q_i(X)$. Further, being an $s$-tuple, $Q_i$ is also a polynomial of degree less than $s$. The polynomial notation facilitates the mathematical developments of codes. The tuple notation is more appropriate for software implementation of codes because an $s$-tuple is a group of $s$ bits, which can be easily processed by computers. Note also that the *ns*-tuple $(Q_0, Q_1, \ldots, Q_{n-2}, Q_{n-1})$ is equivalent to the polynomial $\sum_{i=0}^{n-1} Q_iX^{(n-1-i)s}$ of degree less than $ns$.

First, let $C_1$ be a binary code with length $s$ and minimum distance $d_1$. Let $r$ and $n$ be integers such that $1 \le n \le 2^r$. Let $W_0, W_1, \ldots, W_{n-1}$ be distinct polynomials of degree less than $r$. Let $M$ be a polynomial of degree $r$ such that $M$ and $X$ are relatively prime, i.e., $\gcd(M, X) = 1$. Also, $Q_i$ is an $s$-tuple, $i \ge 0$. Now, we are ready to introduce a new code that is simply called "the code $C$" and is the focus of this paper.

**Algorithm 1.** Let $C$ be the binary code such that each of its codewords

$$(Q_0, Q_1, \ldots, Q_{n-2}, Q_{n-1}) \tag{1}$$

satisfies the following two conditions:

$$\left(\sum_{i=0}^{n-1} Q_iW_i\right) \bmod M = 0 \tag{2}$$

$$\sum_{i=0}^{n-1} Q_i \in C_1. \tag{3}$$

**Remark 1.**

1. $C$ is nonlinear if $C_1$ is nonlinear.
2. From (3), the codewords of $C$ have even weights if the codewords of $C_1$ have even weights.
3. The code $C_1$ in Algorithm 1 can be nonsystematic. However, we focus only on systematic codes, which are more often used in practice. Thus, we assume that $C_1$ is an $(s, s-m, d_1)$ systematic code

with $m$ check bits, $0 \leq m \leq s$. Let $F$ be the encoder of $C_1$. Then, each codeword of $C_1$ is $UX^m + F(U) = (U, F(U))$, where $U$ is an information $(s-m)$-tuple and $F(U)$ is the corresponding check $m$-tuple.

4. In Algorithm 1, the weights $W_0, W_1, \ldots, W_{n-1}$ can be chosen to be distinct polynomials of degree less than $r$ because $1 \leq n \leq 2^r$. However, Algorithm 1 can be extended to allow $n > 2^r$, then $W_0, W_1, \ldots, W_{n-1}$ will not always be distinct (see Section 3 later).
5. All the codes considered in this paper are binary codes, i.e., their codewords consist of digits 0 or 1. In particular, the code $C$ is a binary code whose codewords are $ns$ bits. Computers can efficiently process groups of bits. Thus, as seen in (1), each $ns$-bit codeword is grouped into $n$ tuples, $s$ bits each. Note that this binary code $C$ can also be viewed as a code in GF$(2^s)$, i.e., as a code whose codewords consist of $n$ symbols, each symbol belongs to GF$(2^s)$. More generally, suppose that $ns = xy$ for some positive integers $x$ and $y$, then this same binary code $C$ can also be viewed as a code whose codewords consist of $x$ symbols, each symbol belongs to GF$(2^y)$. In the extreme case ($x = 1$, $y = ns$), the code $C$ is also a code whose codewords consist of only one symbol that belongs to GF$(2^{ns})$. Note that, when the same code is viewed in different alphabets, their respective minimum distances can be very different. For example, consider the binary repetition code $\{0^k, 1^k\}$ of length $k > 1$. When viewed as the binary code over GF(2), this code has minimum distance $d = k$. But, when viewed in GF$(2^k)$, this same code has minimum distance $d = 1$.

Let $d_1$ and $d_C$ be the minimum distances of the binary codes $C_1$ and $C$, respectively, in Algorithm 1. We then have the following theorem that is proven in Appendix A (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm).

**Theorem 1.**

1. $d_C \geq 3$ *if* $d_1 \geq 3$.
2. $d_C = 4$ *if* $d_1 \geq 4$.

**Example 1.** Now, we illustrate Algorithm 1 by constructing a simple binary code $C$. Let $s = 4$, $m = 3$, $r = 2$, and $n = 2^r = 4$. Thus, each codeword of the code $C$ is a 16-tuple $(Q_0, Q_1, Q_2, Q_3)$, where each $Q_i$ is a 4-tuple. Let $M = X^2 + X + 1$ be the modulating (primitive) polynomial. Let the weighting polynomials in (2) be $W_0 = X + 1$, $W_1 = X$, $W_2 = 1$, and $W_3 = 0$. Let $C_1 = \{(0,0,0,0), (1,1,1,1)\}$, i.e., $C_1$ is the $(4,1,4)$ repetition code. Now, we wish to specify the desired codeword $(Q_0, Q_1, Q_2, Q_3)$. Let $Q_0$ and $Q_1$ be two arbitrary 4-tuples. Then, $Q_2$ and $Q_3$ are determined as follows: Let $U_1$ and $U_2$ be arbitrary 2-tuple and 1-tuple, respectively. Then, we define $Q_2 = (U_1, P_1)$ and $Q_3 = (U_2, P_2)$, where $P_1$ and $P_2$ are determined as follows: First, compute the check 2-tuple $P_1 = (Q_0W_0 + Q_1W_1 + U_1X^2) \bmod M$. Next, define

$$\begin{aligned} Y &= Q_0 + Q_1 + (U_1X^2 + P_1) + U_2X^3 \\ &= Q_0 + Q_1 + Q_2 + U_2X^3, \end{aligned}$$

which is a 4-tuple. Thus, $Y$ can be written as $Y = Y_1X^3 + Y_2 = (Y_1, Y_2)$, where $Y_1$ is a 1-tuple and $Y_2$ is a 3-tuple. Finally, we compute $P_2 = Y_2 + (Y_1, Y_1, Y_1)$, which is a 3-tuple.

Now, we will show that the codeword $(Q_0, Q_1, Q_2, Q_3) = (Q_0, Q_1, U_1, P_1, U_2, P_2)$ satisfies (2) and (3) in Algorithm 1. Since $P_1 = (Q_0W_0 + Q_1W_1 + U_1X^2) \bmod M$, we have

$$0 = (Q_0W_0 + Q_1W_1 + U_1X^2 + P_1) \bmod M.$$

Then, $0 = (Q_0W_0 + Q_1W_1 + Q_2W_2 + Q_3W_3) \bmod M$ because $Q_2W_2 = U_1X^2 + P_1$ and $Q_3W_3 = 0$. Thus, $(Q_0, Q_1, Q_2, Q_3)$ satisfies (2). Next,

$$\begin{aligned} Q_0 + Q_1 + Q_2 + Q_3 &= Y + U_2X^3 + Q_3 \\ &\quad (\text{because } Y = Q_0 + Q_1 + Q_2 + U_2X^3) \\ &= Y + U_2X^3 + (U_2, P_2) \\ &= Y + P_2 \; [\text{because } (U_2, P_2) = U_2X^3 + P_2] \\ &= (Y_1, Y_2) + Y_2 + (Y_1, Y_1, Y_1) \\ &= (Y_1, Y_1, Y_1, Y_1) \in C_1. \end{aligned}$$

Thus, $(Q_0, Q_1, Q_2, Q_3) = (Q_0, Q_1, U_1, P_1, U_2, P_2)$ also satisfies (3). By exchanging $P_1$ and $U_2$, the codeword becomes $(Q_0, Q_1, U_1, U_2, P_1, P_2)$, which is a codeword of a systematic code because $(Q_0, Q_1, U_1, U_2)$ are the 11 information bits and $(P_1, P_2)$ are the 5 corresponding check bits. Because $d_1 = 4$, $d_C = 4$ by Theorem 1.2. Thus, $C$ is identical to the $(16, 11, 4)$ extended Hamming code.

## 2.1 Systematic Encoding

In general, the binary code $C$ in Algorithm 1 is not systematic. Now, we construct its systematic versions. Recall that $r$ is the degree of the modulating polynomial $M$ and $s$ is the number of bits contained in each tuple $Q_i$. Let $r \leq s$ and suppose that information tuples

$$(Q_0, Q_1, \ldots, Q_{n-3}, U_1, U_2) \tag{4}$$

are given, where $U_1$ is an $(s-r)$-tuple and $U_2$ is an $(s-m)$-tuple. We wish to append a check $r$-tuple $P_1$ and a check $m$-tuple $P_2$ to (4) so that the resulting codeword is

$$(Q_0, Q_1, \ldots, Q_{n-3}, U_1, U_2, P_1, P_2). \tag{5}$$

Thus, the code $C$ is $ns$ bits long and has $h = r + m$ check bits. Denote $d_C$ as its minimum distance, then $C$ is an $(ns, ns - r - m, d_C)$ code. Then, we have the following algorithm that is proven in Appendix A (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm).

**Algorithm 1a.** When $r \leq s$, the two check tuples of a systematic version of the binary code $C$ can be computed by

$$P_1 = \left( \sum_{i=0}^{n-3} Q_iW_i + U_1X^r \right) \bmod M \tag{6}$$

$$P_2 = Y_2 + F(Y_1), \tag{7}$$

where $W_i \neq 0, 1$ and $F$ is the encoder of $C_1$ as defined in Remark 1.3. The tuples $Y_1$ and $Y_2$ are determined as follows: Let

$$Y = \sum_{i=0}^{n-3} Q_i + U_1 X^r + P_1 + U_2 X^m,$$

which is an $s$-tuple that can be written as $Y = Y_1 X^m + Y_2 = (Y_1, Y_2)$, where $Y_1$ and $Y_2$ are an $(s-m)$-tuple and an $m$-tuple, respectively.

**Remark 2.** After $P_1$ is computed, $P_2$ is easily computed when $C_1$ is one of the following four types of codes: The first two types of codes, given in 1 and 2 below, are very trivial, but they are used later in Section 3 to construct all the codes in Fig. 1. The next two types of codes, given in 3 and 4 below, are commonly used in practice for error control.

1. If $m = s$, then $C_1 = \{0^s\}$, which is an $(s, 0, d_1)$ code, where the minimum distance $d_1$ is undefined. This very trivial code is called a useless code because it carries no useful information. However, it can detect any number of errors, i.e., we can assign $d_1 = \infty$ for this particular code. Further, it can be shown that Theorem 1.2 remains valid when $m = s$, i.e., $d_C = 4$ if $C_1 = \{0^s\}$. Then, from Algorithm 1a, we have $U_2 = 0$, $F = 0^s$, $Y_1 = 0$, and
$$P_2 = Y_2 = Y = \sum_{i=0}^{n-3} Q_i + U_1 X^r + P_1.$$
2. If $m = 0$, then $C_1 = \{0, 1\}^s$, which is an $(s, s, 1)$ code. This very trivial code is called a powerless code because it protects no information. From Algorithm 1a, we have $Y_2 = 0$, $F = 0$,
$$Y_1 = Y = \sum_{i=0}^{n-3} Q_i + U_1 X^r + P_1 + U_2,$$
and $P_2 = 0$.
3. If $C_1$ is a systematic linear code with parity check matrix $\mathbf{H}_1 = [\mathbf{AI}]$, where $\mathbf{A}$ is an $m \times (s-m)$ matrix and $\mathbf{I}$ is the $m \times m$ identity matrix, then $F(U) = U\mathbf{A}^{\mathrm{tr}}$, where "tr" denotes matrix transpose. Thus, $P_2 = Y_2 + F(Y_1) = Y_2 + Y_1 \mathbf{A}^{\mathrm{tr}} = Y\mathbf{H}_1^{tr}$.
4. If $C_1$ is a CRC generated by a polynomial $M_1$ of degree $m$, then $F(U) = (UX^m) \bmod M_1$ (see Appendix B, which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm). Thus,
$$P_2 = Y_2 + (Y_1 X^m) \bmod M_1 = (Y_1 X^m + Y_2) \bmod M_1 = Y \bmod M_1.$$

Algorithm 1a is for the case $r \leq s$, where the check $r$-tuple $P_1$ can be stored in a single $s$-tuple. Now, we consider the case $r > s$. Then, several $s$-tuples are needed to store the check $r$-tuple $P_1$. Because $r > s$, we can write $r = as + b$, where $a \geq 1$ and $0 < b \leq s$.

| $s$ | $r$ | $m$ | $C_1$ | Algo | $a$ | $b$ |
|---|---|---|---|---|---|---|
| 1 | 15 | 1 | $\{0\}$ | 1b | 14 | 1 |
| 2 | 14 | 2 | $\{0^2\}$ | 1b | 6 | 2 |
| 4 | 13 | 3 | (4, 1, 4) | 1b | 3 | 1 |
| 8 | 12 | 4 | (8, 4, 4) | 1b | 1 | 4 |
| 16 | 11 | 5 | (16, 11, 4) | 1a | -- | -- |
| 32 | 10 | 6 | (32, 26, 4) | 1a | -- | -- |
| 64 | 9 | 7 | (64, 57, 4) | 1a | -- | -- |

Fig. 2. Construction of the codes $C$ using16 check bits.

For example, let $s = 8$, then $a = 1$ and $b = 4$ if $r = 12$, whereas $a = 1$ and $b = 8$ if $r = 16$. Thus, $P_1$ can be stored in $a + 1$ tuples: The first tuple is $b$ bits and each of the next $a$ tuples is $s$ bits. Now, assume that information tuples $(Q_0, Q_1, \ldots, Q_{n-a-3}, U_1, U_2)$ are given, where each $Q_i$ is $s$ bits, $U_1$ is $s - b$ bits, and $U_2$ is $s - m$ bits. We assume here that $n - a - 3 \geq 0$ or $n \geq a + 3$, to avoid triviality. We wish to append two checks tuples $P_1$ and $P_2$ to $(Q_0, Q_1, \ldots, Q_{n-a-3}, U_1, U_2)$ so that

$$(Q_0, Q_1, \ldots, Q_{n-a-3}, U_1, U_2, P_1, P_2)$$

becomes a codeword of a systematic $(ns, ns - r - m, d_C)$ code. Then, we have the following algorithm that is proven in Appendix A (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm).

**Algorithm 1b.** When $r > s$, the two check tuples of a systematic version of the binary code $C$ can be computed by

$$P_1 = \left( \sum_{i=0}^{n-a-3} Q_i W_i + U_1 X^r \right) \bmod M \text{ and } P_2 = Y_2 + F(Y_1),$$

where $F$ is the encoder of $C_1$ and

$$W_i \neq X^{as}, X^{(a-1)s}, \ldots, X^s, 1, 0.$$

The tuples $Y_1$ and $Y_2$ are determined as follows: Define

$$Y = \left( \sum_{i=0}^{n-a-3} Q_i \right) + \left( U_1 X^b + P_{10} \right) + \left( \sum_{i=1}^{a} P_{1i} \right) + U_2 X^m,$$

where $P_{10}$ is a $b$-tuple and $P_{11}, \ldots, P_{1a}$ are $s$-tuples that satisfy $P_1 = (P_{10}, P_{11}, \ldots, P_{1a})$. Then, $Y$ is an $s$-tuple that can be written as $Y = Y_1 X^m + Y_2 = (Y_1, Y_2)$, where $Y_1$ and $Y_2$ are an $(s-m)$-tuple and an $m$-tuple, respectively.

**Example 2.** Recall that $C$ is an $(ns, ns - r - m, d_C)$ code that is constructed by either Algorithm 1a (if $r \leq s$) or Algorithm 1b (if $r > s$). This code has $h = r + m$ check bits. In this example, we assume that $h = 16$ bits and we present different ways to construct the codes $C$. The results are shown in Fig. 2. For example, using Algorithm 1b, we can construct the code $C$ with the following parameters: $s = 8$, $r = 12$, $m = 4$, $C_1 = (8, 4, 4)$ code, $a = 1$, and $b = 4$ ($a$ and $b$ are not needed in Algorithm 1a). Assume that the number of $s$-tuples satisfies $n \leq 2^r$, i.e., the number of bits in each codeword is $ns \leq 2^r s = 2^{15}$. Then, the weighting polynomials $W_i$ can be chosen to be distinct. From Remark 2.1, we have $d_1 \geq 4$. Then, from Theorem 1.2, all the codes $C$ in Fig. 2 have minimum distance $d_C = 4$.

## 3 Some Special Error-Detection Codes

This section shows that the binary code $C$ of Algorithm 1 is general in the sense that it includes all the codes in Fig. 1 and other codes as special cases. Recall that Algorithm 1's systematic version is either Algorithm 1a (if $r \leq s$) or Algorithm 1b (if $r > s$), where $r$ is the degree of the modulating polynomial $M$ and $s$ is the number of bits contained in each tuple $Q_i$. The code $C$ depends on the components such as the parameters $r, m, s, n$, the weights $W_0, W_1, \ldots, W_{n-1}$, and the code $C_1$. Thus, different components will produce different codes $C$. We now show that Algorithm 1 produces the codes in Fig. 1 by letting $C_1$ be trivial codes such as $(s, s, 1)$ and $\{0^s\}$ defined in Remark 2. The algorithm also produces other linear and nonlinear codes (Sections 3.1, 3.6, and 3.8). Generally, codes of $d_C = 4$ require that $n \leq 2^r$ and the weights $W_0, W_1, \ldots, W_{n-1}$ in Algorithm 1 be distinct. Codes of $d_C = 3$ require that $n \leq 2^r + 1$ and allow some of the weights to be repeated. Codes of $d_C = 2$ also allow some of the weights to be repeated, but do not restrict on the value of $n$, i.e., the code lengths can be arbitrary. The following codes are briefly presented because their detailed discussions can be found elsewhere [4], [5], [9], [14].

### 3.1 Binary Extended Perfect Code

We now show that Algorithm 1 produces an extended perfect code if the code $C_1$ is an extended perfect code. Suppose that $C_1$ is a $(2^{m-1}, 2^{m-1} - m, 4)$ extended perfect code (see [8, Chapter 6]), i.e., $s = 2^{m-1}$ and $d_1 = 4$. Let $n = 2^r$ and $h = r + m$, then the code $C$ has $ns = 2^{r+m-1} = 2^{h-1}$ bits. Then, $d_C = 4$ by Theorem 1.2 and $C$ is a $(2^{h-1}, 2^{h-1} - h, 4)$ extended perfect code. Note that deleting a check bit from an extended perfect code will yield a perfect code, while adding an overall even parity bit to a perfect code will yield an extended perfect code.

Algorithms 1a and 1b can be further generalized to include the extended perfect code of [15] as follows: Recall that $P_1$, $P_2$, and $Y_1$ are the check $r$-tuple, check $m$-tuple, and $(s-m)$-tuple, respectively, which are computed from Algorithms 1a or 1b. Let $E(.)$ be any function from the set of $(s-m)$-tuples to the set of $r$-tuples. Now, define the new check $r$-tuple and check $m$-tuple by

$$P_1^* = P_1 + E(Y_1) \text{ and } P_2^* = P_2 + \text{ even parity of } E(Y_1).$$

Then, it can be shown that, if $C_1$ is an extended perfect code and $n = 2^r$, then the resulting code $C$ whose check tuples are $P_1^*$ and $P_2^*$ is also an extended perfect code. Further, when $r = 1$, this extended perfect code becomes the extended perfect code that is obtained from the systematic perfect code of [15].

### 3.2 Weighted Sum Code (WSC)

Consider the code $C$ for the special case $s = r = m$. By Remark 2.1, we have $C_1 = \{0^s\}$, $U_1 = 0$, $U_2 = 0$, $Y_1 = 0$, and $Y_2 = Y = \sum_{i=0}^{n-3} Q_i + P_1$. From (6) and (7) of Algorithm 1a, we have

$$P_1 = \sum_{i=0}^{n-3} Q_i W_i \bmod M \text{ and } P_2 = \sum_{i=0}^{n-3} Q_i + P_1. \tag{8}$$

Thus, this special code $C$ is the WSC presented in [4], [9]. It is shown in [3] that the WSC, when viewed as a code in GF$(2^s)$, is equivalent to a lengthened single-error correcting Reed Solomon code (see also [8, p. 323]).

### 3.3 Block-Parity Code

Suppose that $r = 0$ and $m = s$. Thus, by Remark 2.1, $C_1 = \{0^s\}$, $Q_{n-2} = U_1$, $P_1 = 0$ (because $r = 0$), $Y_1 = 0$, and $U_2 = 0$ (because $m = s$). Then,

$$Y_2 = Y = \sum_{i=0}^{n-3} Q_i + U_1 = \sum_{i=0}^{n-2} Q_i.$$

From (7) of Algorithm 1a, we have $P_2 = Y = \sum_{i=0}^{n-2} Q_i$. Thus, the resulting code $C$ is the block-parity code presented in [4].

### 3.4 Cyclic Redundancy Code (CRC)

Consider an $h$-bit CRC that is $q$ bits long and is generated by a polynomial $M$. Suppose that $q$ and $h$ can be written as $q = x + (n-1)s$ and $h = as + b$, where $n \geq 1$, $0 \leq x \leq s$, $a \geq 0$, and $0 < b \leq s$ (see Appendix B, which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm). Then, it is shown in Remark B1 that the CRC check tuple is

$$P = \left( \sum_{i=0}^{n-a-2} Q_i W_i + U_1 X^r \right) \bmod M,$$

where $W_i = X^{(n-1-i)s} \bmod M, i = 0, 1, \ldots, n-a-2$. Further, we show in Remark B1 that the weighting polynomials $W_i$ are distinct and $W_i \neq 0, 1, X^s, \ldots, X^{as}$, provided that $q \leq 2^{h-1} - 1$ and $M$ is the product of $(X+1)$ and a primitive polynomial of degree $h-1$.

Now, consider the code $C$ that has the same length and the same weighting polynomials as the above CRC. Let $r = h$ and $m = 0$. Then, $P_2 = 0$ by Remark 2.2 and $P_1 = P$ by Algorithm 1a (if $r \leq s$) or by Algorithm 1b (if $r > s$). Thus, this particular code $C$ is identical to the above CRC. So, any CRC can be generated by either Algorithm 1a or Algorithm 1b, i.e., by Algorithm 1.

**Remark 3.** To construct other codes (such as CXOR checksum and nonbinary Hamming codes), we need to modify (3) by deleting $Q_{n-2}$ from the summation, but (2) remains unchanged. That is, (3) is replaced by

$$\left( \sum_{i=0}^{n-3} Q_i + Q_{n-1} \right) \in C_1. \tag{9}$$

Then, Algorithm 1a remains valid if we define $Y = \sum_{i=0}^{n-3} Q_i + U_2 X^m$ because the term $Q_{n-2} = U_1 X^r + P_1$ is absent from (9).

### 3.5 CXOR Checksum

Suppose now that we allow some of the polynomials $W_0, W_1, \ldots, W_{n-1}$ in (2) to be repeated and we use Algorithm 1a along with variation (9). Let $r = s = m$, $M = X^s + 1$, and $W_i = X^i \bmod M$. It can be shown that $W_{i+s} = W_i$ for all $i \geq 1$, i.e., some of the weighting polynomials may repeat. Then, $C_1 = \{0^s\}$ (because $m = s$), $U_1 = 0$ (because $r = s$), and $U_2 = Y_1 = 0$ (because $m = s$). From (6) and (7), we have $P_1 = \sum_{i=0}^{n-3} Q_i X^i \bmod (X^s + 1)$ and $P_2 = Y_2 = Y = \sum_{i=0}^{n-3} Q_i$ (see Remark 2.1 and Remark 3). Thus, the resulting code $C$ is the CXOR checksum presented in [4].

## 3.6 Nonbinary Perfect Code

Suppose that Algorithm 1a along with variation (9) is applied with $r = s = m$ and $n = 2^m + 1$. Let $M$ be a primitive polynomial of degree $m$ and let

$$W_0, W_1, \ldots, W_{n-3}$$

be distinct and nonzero polynomials. Then, $C_1 = \{0^s\}$, $P_1 = \sum_{i=0}^{2^m-2} Q_i W_i \bmod M$, and $P_2 = \sum_{i=0}^{2^m-2} Q_i$. It then can be shown that $P_1$ and $P_2$ are two check tuples of the nonbinary Hamming perfect code over GF$(2^m)$ (see [8, Chapter 6]), i.e., the tuples $(Q_0, Q_1, \ldots, Q_{2^m-2}, P_1, P_2)$ form the codewords of the Hamming perfect code over GF$(2^m)$.

## 3.7 One's-Complement Checksum and Fletcher Checksum

The above codes are constructed using polynomial arithmetic because each tuple is considered as a polynomial over the binary field $\{0, 1\}$. An alternative is to consider each tuple as an integer and to use the rules of (one's-complement) integer arithmetic to manipulate the code construction. If we apply the integer arithmetic to the construction of the block-parity code and to the nonbinary perfect code, we will get the one's-complement checksum and Fletcher checksum, respectively. However, these integer-based codes are often weaker than their binary polynomial counterparts (see Fig. 1). See [4], [5], [9] for definitions and performance comparisons of error-detection codes, including the one's-complement and Fletcher checksums. Thus, the integer-arithmetic version of Algorithm 1a, along with variation (9), also produces the one's-complement and Fletcher checksums. We will not discuss these checksums and integer-based codes any further because they are often weaker than their polynomial counterparts and their analyses can be found elsewhere (e.g., [5], [14]).

## 3.8 Other Error-Detection Codes

Recall from Algorithms 1a and 1b that the $(ns, ns - r - m, d_C)$ code $C$ is constructed from an $(s, s - m, d_1)$ code $C_1$. Thus, by varying $C_1$, different codes $C$ are produced. Further, $C$ is nonlinear if $C_1$ is nonlinear. Thus far, the codes $C$ are constructed from the codes $C_1$ that are either extended perfect codes or trivial codes $\{0^s\}$ and $(s, s, 1)$. Now, we construct the codes $C$ from the codes $C_1$ that are neither perfect nor trivial. In both instances below, we assume that $s = r + m = 16$, $n = 2^r$ with $r = 7$ or 8, and $d_1 = 6$, so that $d_C = 4$ by Theorem 1.2.

1. Suppose that $C_1$ is the extended $(16, 7, 6)$ linear BCH code (see [8], Chapter 3) and $r = 7$. Then, $ns = 2^r s = 2{,}048$ and the resulting code $C$ is a $(2{,}048, 2{,}032, 4)$ linear code.
2. Suppose that $C_1$ is the extended $(16, 8, 6)$ nonlinear Nordstrom-Robinson code (see [8, p. 73]) and $r = 8$. Then, $ns = 2^r s = 4{,}096$, and $C$ is a $(4{,}096, 4{,}080, 4)$ nonlinear code that is twice as long as the linear code in 1.

# 4 Fast Implementation of Error-Detection Codes

Recall from Algorithm 1 that $r$ is the degree of the modulating polynomial $M$ and $s$ is the number of bits contained in each tuple $Q_i$. Algorithm 1 produces a large family of error-detection codes because its systematic versions (either Algorithm 1a when $r \le s$ or Algorithm 1b when $r > s$) generate all the codes presented in Section 3. So far, the discussion is abstract and general to facilitate the development of mathematical properties of our algorithms. In this section, we focus on the practical aspect of these algorithms, i.e., we now discuss how some codes generated by these algorithms can be *efficiently* implemented in software. Then, we compare the complexity of our algorithms with that of the CRC algorithm (the strongest code in Fig. 1). In theory, the fundamental unit for digital data is *bit*. In practice, however, communication protocols and computers often process data as blocks of bits or tuples (e.g., bytes or words) and not as individual bits at a time. For example, on familiar 32-bit computers, the modulo-2 addition of two 32-bit numbers can be accomplished by a single XOR operation (using C programming language). Thus, efficient error-detection codes should also be processed in terms of tuples at a time, i.e., each *ns*-bit codeword is expressed in terms of $n$ tuples, $s$ bits each.

In parallel to Algorithm 1a and Algorithm 1b, now we develop two fast algorithms: Algorithm 2a for $r \le s$ and Algorithm 2b for $r > s$. Although Algorithms 1a and 1b can produce CRCs and many other codes (see Section 3), the two fast algorithms produce only non-CRC codes that are shown later in Section 4.1 to be faster than CRCs by the factor O$(s)$.

Now, suppose that information tuples

$$(Q_0, Q_1, \ldots, Q_{n-3}, U_1, U_2)$$

are given. Let $r$, $s$, and $m$ be such that $r \le s$ and $n \le 2^r$. Assume that each $Q_i$ is $s$ bits, $U_1$ is $s - r$ bits, and $U_2$ is $s - m$ bits. From the following algorithm, we can compute the two check tuples $P_1$ and $P_2$ that are appended to the information tuples such that the resulting code $C$ has minimum distance $d_C = 4$.

**Algorithm 2a.** Let $r \le s$ and $n \le 2^r$. Let $M$ be a primitive polynomial of degree $r$ and let $F$ be the encoder of an $(s, s - m, d_1)$ code with $d_1 \ge 4$. Then, the resulting code $C$ is an $(ns, ns - r - m, 4)$ code and each of its codewords is $(Q_0, Q_1, \ldots, Q_{n-3}, U_1, U_2, P_1, P_2)$. The two check tuples are computed by

$$P_1 = (Z + U_1 X^r) \bmod M \text{ and } P_2 = Y_2 + F(Y_1),$$

where $Z = \sum_{i=0}^{n-3} Q_i X^{n-2-i} \bmod (M X^{s-r})$. The tuples $Y_1$ and $Y_2$ are defined as in Algorithm 1a, i.e., they satisfy $(Y_1, Y_2) = Y_1 X^m + Y_2 = Y = \sum_{i=0}^{n-3} Q_i + U_1 X^r + P_1 + U_2 X^m$.

**Proof.** Define $W_i = X^{n-2-i} \bmod M$, $i = 0, 1, \ldots, n-3$. Then, $W_i \ne 0, 1$ and $W_0, W_1, \ldots, W_{n-3}$ are distinct because $M$ is a primitive polynomial of degree $r$ and $n \le 2^r$. Let $C_1$ be the $(s, s - m, d_1)$ code with the encoder $F$. Now, using these $W_i$ and $C_1$ in Algorithm 1a, we can construct the code $C$ whose two check tuples are given by

$$P_1 = \left( \sum_{i=0}^{n-3} Q_i W_i + U_1 X^r \right) \bmod M \quad \text{and} \quad P_2 = Y_2 + F(Y_1).$$

Because $d_1 \geq 4$, $d_C = 4$ by Theorem 1.2. Next, the new form of $P_1$ is derived as follows: First, note that

$$X^{s-r} W_i = (X^{s-r} X^{n-2-i}) \bmod (M X^{s-r}). \qquad (10)$$

Multiplying $P_1$ by $X^{s-r}$, we have

$$P_1 X^{s-r} = \left( \sum_{i=0}^{n-3} Q_i W_i + U_1 X^r \right) X^{s-r} \bmod (M X^{s-r}). \qquad (11)$$

From (10), (11), the definition of $Z$, and some modular algebra manipulation, it can be shown that $P_1 X^{s-r} = (X^{s-r} Z + U_1 X^r X^{s-r}) \bmod (M X^{s-r})$. Thus,

$$P_1 X^{s-r} = ((Z + U_1 X^r) \bmod M) X^{s-r}. \qquad (12)$$

From (12), we have $P_1 = (Z + U_1 X^r) \bmod M$. $\square$

**Remark 4.** Let $I$ and $J$ be two binary polynomials of degrees $i$ and $j$, respectively. In general, it is rather tedious to compute $I \bmod J$. However, when $i \leq j$, the computation becomes easy and is accomplished in constant time because

$$I \bmod J = \begin{cases} I & \text{if } i < j \\ I + J & \text{if } i = j. \end{cases}$$

This simple fact is used to efficiently compute $Z = \sum_{i=0}^{n-3} Q_i X^{n-2-i} \bmod (M X^{s-r})$ in Algorithm 2a, as follows: Using Horner's rule, we have

$$Z = (\ldots((Q_0 X) \bmod N + Q_1) X \bmod N + \ldots + Q_{n-3}) \bmod N,$$

where $N = M X^{s-r}$. Then, $Z$ can be recursively computed from the polynomials $T_i$ defined by $T_0 = Q_0$ and $T_i = (T_{i-1} X) \bmod N + Q_i$, $i = 1, \ldots, n-3$. Because $s = \deg N \geq \deg(T_{i-1} X)$, each $T_i$ is computed in constant time, i.e., with O(1) complexity. Finally, we have $Z = (T_{n-3} X) \bmod N$. Thus, $Z$ has computational complexity $O(n)$. Horner's rule is also used to efficiently encode the WSC [4], [9].

Fig. 3 shows a simple software implementation of Algorithm 2a. The input data are $(Q_0, Q_1, \ldots, Q_{n-3}, U_1, U_2)$. The output is the check tuple $P = (P_1, P_2)$. The "for" loop is used to compute both $Y$ and $T$. Computation of $Y$ requires only one XOR operation, while $T$ can be efficiently computed via Remark 4 because $\deg N \geq \deg(TX)$. Then, the final value of $T$ is used to compute $Z$, i.e., $Z = (TX) \bmod N$.

Now, we consider a fast version of Algorithm 1b for constructing the code $C$. In this case, $r > s$, i.e., $r = as + b$, where $a \geq 1$ and $0 < b \leq s$. Assume that information tuples $(Q_0, Q_1, \ldots, Q_{n-a-3}, U_1, U_2)$ are given, where each $Q_i$ is $s$ bits, $U_1$ is $s - b$ bits, and $U_2$ is $s - m$ bits. We wish to append two checks tuples $P_1$ and $P_2$ to the information tuples so that $(Q_0, Q_1, \ldots, Q_{n-a-3}, U_1, U_2, P_1, P_2)$ is a codeword of the code $C$. Before stating the algorithm, we need some preliminary results.

```
$N = MX^{s-r}$;
$T = Q_0; Y = Q_0$;
for $(1 \leq i \leq n-3)$
  {
    $T = (TX) \bmod N + Q_i$;
    $Y = Y + Q_i$;
  }
$Z = (TX) \bmod N$;
$P_1 = (Z + U_1X^r) \bmod M$;
$Y = Y + U_1X^r + P_1 + U_2X^m$;
$Y = Y_1X^m + Y_2$;
$P_2 = Y_2 + F(Y_1)$;
$P = P_1X^m + P_2$;
```

Fig. 3. Pseudocode for Algorithm 2a. Here, $Q_i$, $U_1$, and $U_2$ are input information tuples, $P$ is the output check tuple.

**Remark 5.** Algorithm 2b, which will be discussed shortly, requires operations on new tuples $Q_0^*, Q_1^*, \ldots, Q_{n-3}^*$ that are defined from the original tuples $Q_0, Q_1, \ldots, Q_{n-a-3}$ as follows: First, let $\mathbf{U} = \{0, 1, \ldots, n-4, n-3\}$, then we partition the set $\mathbf{U}$ into four sets $\mathbf{P}$, $\mathbf{Q}$, $\mathbf{X}$, and $\mathbf{Y}$:

$$\begin{aligned} \mathbf{P} &= \{i : 0 \leq i \leq n-3-a, i = n-2-js \\ &\quad \text{for some } j, 1 \leq j \leq a\}, \\ \mathbf{Q} &= \{i : 0 \leq i \leq n-3-a, i \neq n-2-js \\ &\quad \text{for all } j, 1 \leq j \leq a\}, \\ \mathbf{X} &= \{i : n-3-a < i \leq n-3, i = n-2-js \\ &\quad \text{for some } j, 1 \leq j \leq a\}, \\ \mathbf{Y} &= \{i : n-3-a < i \leq n-3, i \neq n-2-js \\ &\quad \text{for all } j, 1 \leq j \leq a\}. \end{aligned}$$

Because $|\mathbf{P}| + |\mathbf{X}| \leq a$ and $a = |\mathbf{X}| + |\mathbf{Y}|$, we have $|\mathbf{P}| \leq |\mathbf{Y}|$. Let $p = |\mathbf{P}|$, then $\mathbf{Y}$ has at least $p$ elements. So, let $\mathbf{Y}^*$ be the set of $p$ smallest elements of $\mathbf{Y}$, i.e., $\mathbf{Y}^* = \{f_1, f_2, \ldots, f_p\}$. Similarly, we can write $\mathbf{P} = \{e_1, e_2, \ldots, e_p\}$. Finally, we can define the new tuples $Q_0^*, Q_1^*, \ldots, Q_{n-3}^*$ from $Q_0, Q_1, \ldots, Q_{n-a-3}$ as follows:

$$Q_i^* = \begin{cases} 0 & \text{if } i \in \mathbf{P} \cup \mathbf{X} \\ Q_i & \text{if } i \in \mathbf{Q} \\ 0 & \text{if } p < i \leq n-3, \end{cases}$$

and $Q_{f_i}^* = Q_{e_i}$ if $1 \leq i \leq p$.

**Remark 6.** Now, assume that $s \geq a + 1$ and $n \geq 2 + as$, we will show that these conditions will simplify the definition of $Q_i^*$ (given in Remark 5). It can be shown from these conditions that $0 \leq n - 2 - js \leq n - 3 - a$ for all $1 \leq j \leq a$. Then, from Remark 5, we have $p = |\mathbf{P}| = a$, $|\mathbf{X}| = 0$, $|\mathbf{Y}| = a$, and $\mathbf{Y}^* = \mathbf{Y}$. We also have $\mathbf{Y}^* = \{n-2-j, j = a, a-1, \ldots, 1\}$ and $\mathbf{P} = \{n-2-js, j = a, a-1, \ldots, 1\}$. Note that $f_i = n - 2 - (a+1-i)$ and $e_i = n - 2 - (a+1-i)s$, $1 \leq i \leq a$. Thus, $Q_{f_i}^* = Q_{e_i}$ iff $Q_{n-2-(a+1-i)}^* = Q_{n-2-(a+1-i)s}$ iff $Q_{n-2-js}^* = Q_{n-2-js}$. Finally, from Remark 5, we have

$Q_i^* = Q_i$ if $i \neq n-2-js, 1 \leq j \leq a, 0 \leq i \leq n-3-a,$
$Q_{n-2-js}^* = 0, 1 \leq j \leq a,$ and
$Q_{n-2-j}^* = Q_{n-2-js}, 1 \leq j \leq a.$

Basically, $Q_0^*, Q_1^*, \dots, Q_{n-3}^*$ are obtained by moving some $a$ tuples of $Q_0, Q_1, \dots, Q_{n-a-3}$ to the right and then by filling the $a$ removed tuples by zeros. Now, define $Z = \sum_{i=0}^{n-3} Q_i^* X^{n-2-i} \bmod M$, which is a key quantity in the following algorithm. This simplified definition of $Q_i^*$ makes it possible to calculate $Z$ directly from $Q_i$ (i.e., without using $Q_i^*$). That is, we first modify $Q_i$ using the following pseudocode:

```
for($1 \leq j \leq a$)$Q_{n-2-j} = Q_{n-2-js}$;
for($1 \leq j \leq a$)$Q_{n-2-js} = 0$;
```

then we can compute $Z$ directly from the modified $Q_i$ as $Z = \sum_{i=0}^{n-3} Q_i X^{n-2-i} \bmod M$.

Now, we have the following algorithm that is proven in Appendix A (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm).

**Algorithm 2b.** Suppose that $r > s$ and $n \leq 2^r$. Let $M$ be a primitive polynomial of degree $r$ and let $F$ be the encoder of an $(s, s-m, d_1)$ code with $d_1 \geq 4$. Then, the two check tuples of the code $C$ are computed by

$$P_1 = (Z + U_1 X^r) \bmod M \text{ and } P_2 = Y_2 + F(Y_1),$$

where $Z = \sum_{i=0}^{n-3} Q_i^* X^{n-2-i} \bmod M$ and $Q_i^*$ are defined in Remark 5 (or in Remark 6 if applicable). The tuples $Y_1$ and $Y_2$ are defined as in Algorithm 1b, i.e., they satisfy

$$\begin{aligned}(Y_1, Y_2) &= Y_1 X^m + Y_2 = Y \\ &= \left(\sum_{i=0}^{n-a-3} Q_i\right) + \left(U_1 X^b + P_{10}\right) + \left(\sum_{i=1}^{a} P_{1i}\right) + U_2 X^m,\end{aligned}$$

where $P_{10}$ is a $b$-tuple, and $P_{11}, \dots, P_{1a}$ are $s$-tuples that satisfy $P_1 = (P_{10}, P_{11}, \dots, P_{1a})$. Further, $C$ is an $(ns, ns - r - m, 4)$ code.

Fig. 4 shows a software implementation of Algorithm 2b under the assumption that $s \geq a+1$ and $n \geq 2 + as$ as required in Remark 6. The input data are $Q_0, Q_1, \dots, Q_{n-a-3}, U_1, U_2$. The output is the check tuple $P = (P_1, P_2)$. Note that, as in Algorithm 2a (see Remark 4), the tuple $Z$ in Algorithm 2b can also be computed in time $O(n)$.

**Example 3.** Here, we construct the code $C$ for the case $r > s$, with $s = 8$ and $r = 12$. Let $m = 4$, then the total number of check bits is $h = r + m = 16$. Because $r > s$, we can write $r = as + b$ with $a = 1$ and $b = 4$. Let $(Q_0, Q_1, \dots, Q_{n-4}, U_1, U_2)$ be information tuples, where each $Q_i$ is an 8-tuple, $U_1$ and $U_2$ are 4-tuples, which can be combined into a single 8-tuple $(U_1, U_2)$. We wish to append a check 16-tuple $(P_1, P_2)$ to the information tuples so that $(Q_0, Q_1, \dots, Q_{n-4}, (U_1, U_2), P_1, P_2)$ forms a codeword of the code $C$, which is an $(8n, 8n-16, 4)$ code. Here, we let $F$ be the encoder of the $(8, 4, 4)$ extended Hamming code. The resulting code $C$ can have length up to $2^{h-1} = 2^{15}$ bits (see Section 3.1).

```
for ($1 \leq i \leq a$) $Q_{n-2-i} = Q_{n-2-is}$;
for ($1 \leq i \leq a$) $Q_{n-2-is} = 0$;
$T = Q_0$; $Y = Q_0$;
for ($1 \leq i \leq n-3$)
  {
  $T = (TX) \bmod M + Q_i$;
  $Y = Y + Q_i$;
  }
$Z = (TX) \bmod M$;
$P_1 = (Z + U_1 X^r) \bmod M$;
$P_1 = (P_{10}, P_{11}, \dots, P_{1a})$;
$P_2 = P_{10} + U_1 X^b$;
for ($1 \leq i \leq a$) $P_2 = P_2 + P_{1i}$;
$Y = Y + P_2 + U_2 X^m = Y_1 X^m + Y_2$;
$P_2 = Y_2 + F(Y_1)$;
$P = P_1 X^m + P_2$;
```

Fig. 4. Pseudocode for Algorithm 2b. Here, $Q_i$, $U_1$, and $U_2$ are input information tuples, $P$ is the output check tuple.

In this example, $a = 1$, $s = 8$, and $n$ is the total number of bytes in a codeword of the code $C$. If we assume further that $n \geq 10$, then the hypotheses of Remark 6 are satisfied, i.e., $s \geq a+1$ and $n \geq 2 + as$. Thus, by Remark 6, we can modify the $Q_i$ by first setting $Q_{n-3} = Q_{n-10}$ and then setting $Q_{n-10} = 0$. Then, the modified information tuples are

$$(Q_0, Q_1, \dots, Q_{n-11}, Q_{n-10}, Q_{n-9}, \dots, Q_{n-4}, Q_{n-3}, (U_1, U_2)).$$

Then, as in Algorithm 2a, we can efficiently compute the quantity $Z = \sum_{i=0}^{n-3} Q_i X^{n-2-i} \bmod M$, which is shown in Fig. 4.

**Remark 7.**

1. Given $r$ and $s$, either Algorithm 2a or Algorithm 2b can be used to construct the code $C$ that is $ns$ bits long, where $1 \leq n \leq 2^r$. The values of $r$ and $s$ can be as small as 0 and 1, respectively. However, the resulting code $C$ can be trivial, e.g., if $r = 0$, then $n = 1$ and $C = C_1$. If $r = 0$, $s = 1$, and $C_1 = \{0\}$, then $C = C_1 = \{0\}$. If $s = 1$, $C_1 = \{0\}$, $r = 1$, and $n = 2^r = 2$, then $C = \{(0,0)\}$. However, when $s = 1$, $C_1 = \{0\}$, $r \geq 2$, and $n \geq 4$, the resulting code $C$ can be nontrivial and each codeword of $C$ now has $ns = n$ bits. In particular, from Algorithm 2b, it can be shown that the two check tuples of an $n$-bit codeword are

$$P_1 = Z \text{ and } P_2 = \sum_{i=0}^{n-a-3} Q_i + \sum_{i=0}^{a} P_{1i},$$

i.e., $P_2$ is the even parity bit computed from the first $n-1$ bits of the codeword of $C$. For example, if $r = 2$ and $n = 2^r = 4$, then $C$ is the $(4, 1, 4)$ repetition code. This $(4, 1, 4)$ code is also constructed from Algorithm 2a with $r = 1$, $n = 2^r = 2$, $s = 2$, and $C_1 = \{(0, 0)\}$.

2. Let $r = 1$, then $M = X + 1$. Thus, the code $C$ is $2s$ bits long (by Algorithm 2a) and $P_1 = (U_1X) \bmod (X + 1)$, which is the even parity of $U_1$. For example, let $C_1$ be the $(4, 1, 4)$ code, then we can construct the code $C$ of length 8, which is the $(8, 4, 4)$ extended Hamming code. If we set $C_1 = (8, 4, 4)$ code, then we can construct the code $C$ of length 16, i.e., $C = (16, 11, 4)$ code. Repeating this process, we can construct $(32, 26, 4)$ and $(64, 57, 4)$ codes. This method is related to the scheme of [15] and is effective to construct codes that are small enough to fit into the computer words.
3. Let $r \geq 0$, $s \geq 1$, $C_1 = (s, s - m, d_1)$ with $d_1 \geq 4$, and $h = r + m$. Then, using either Algorithm 2a (if $r \leq s$) or Algorithm 2b (if $r > s$), we can construct the code $C$ that is an $(ns, ns - h, 4)$ code. In particular, if $n = 2^r$, then $C$ is a $(2^r s, 2^r s - h, 4)$ code. That is, starting from a code $C_1$ of length $s$, we can construct the code $C$ of length $2^r s$. Further, if $C_1$ is a $(2^{m-1}, 2^{m-1} - m, 4)$ extended perfect code, then $C$ is a $(2^{h-1}, 2^{h-1} - h, 4)$ extended perfect code. If $C_1$ is a linear perfect code, then $C$ is also a linear perfect code. This linear perfect code $C$ and the extended Hamming perfect code of length $2^{h-1}$ are equivalent, i.e., one code can be obtained from the other code by reordering the bit positions and adding a constant vector (see [8, p. 39]). Equivalent codes have the same minimum distance and length, but their implementation complexity can be very different. However, our algorithms can also generate fast codes that are different from the perfect codes. For example, in Algorithm 2a, let $s = 16$ and let $F$ be the encoder of the extended $(16, 8, 6)$ nonlinear Nordstrom-Robinson code (see also Section 3.8). Then, the resulting code $C$ is a nonlinear code with $d_C = 4$, which is not equivalent to any extended perfect codes.

## 4.1 Software Complexity

Now, we compare software complexity between the code $C$ and the CRC (the strongest code in Fig. 1). Here, we focus on implementations that require no table lookup. Table-lookup methods are discussed later in Remark 8.2

Suppose that $s \geq r$. Then, the binary code $C$ of length $ns$ bits can be constructed using Algorithm 2a whose complexity is dominated by the computation of $Z$ and $Y$, which can be computed by the for-loop in Fig. 3. Within this for-loop, the expression $T = (TX) \bmod N + Q_i$ is computed in constant time (by Remark 4), while the expression $Y = Y + Q_i$ is computed by one XOR operation. Thus, this for-loop has complexity $\mathrm{O}(n)$. Hence, the time complexity of the code $C$ is also $\mathrm{O}(n)$. Similarly, when $s < r$, the code $C$ under Algorithm 2b also has time complexity $\mathrm{O}(n)$ (see Fig. 4). In summary, regardless of $s \geq r$ or $s < r$, the code $C$ of length $ns$ can be encoded with time complexity $\mathrm{O}(n)$.

Now, consider the CRC that also has length $ns$ bits. Here, we limit our discussions to a generic CRC algorithm, i.e., a general algorithm that is applicable to all generating polynomials. Then, it is shown in Remark B3(a) that the generic CRC algorithm has time complexity $\mathrm{O}(ns)$. For some specific generating polynomials whose nonzero terms satisfy certain desirable properties, alternative algorithms (such as shift and add [4] and on-the-fly [11]) may have lower complexity.

When $s$ is considered as a constant, we have $\mathrm{O}(ns) = \mathrm{O}(n)$. Thus, from a purely theoretical viewpoint, both the CRC and the code $C$ have the same level of complexity. However, the extra factor $s$ does not appear in the time complexity of the code $C$, i.e., the code $C$ is approximately faster than the CRC by the factor $\mathrm{O}(s)$. We will show later, in Remark 8.1, that $\mathrm{O}(s) \approx 0.73s$ when these error-detection codes are implemented in C programming language.

**Example 4.** Here, we study codes of $h = 16$ check bits (other values of $h$ are discussed later in Remark 8.1). Assume that $C_1$ is the $(s, s - m, 4)$ extended Hamming code and the resulting code $C$ is constructed by Algorithm 2a or Algorithm 2b. Thus, both the CRC and the code $C$ have minimum distance $d = 4$ and the maximum code lengths of the code $C$ and of the CRC are $2^{15}$ and $2^{15} - 1 \approx 2^{15}$ bits, respectively (see also Remark 7.3). Thus, in terms of the minimum distance and maximum code length, the code $C$ and the CRC perform almost identically. Our goal here is to compare the software complexity of these two codes. Software complexity refers to the number of software operations to process one byte of a codeword. Here, a code is called "faster" if it has lower operation count. Simply stated, we write software programs (in C programming language) for the code $C$ and the CRC. Then, we count the number of software operations needed by each code to encode one byte of a codeword. Computer programs for these codes and the rules for counting the operations are given in Appendix D (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm).

Recall that a typical codeword consists of $n$ tuples, each tuple has $s$ bits. Let $t_C(s, n)$ and $t_{CRC}(s, n)$ be the software operation count required to compute the $h = 16$ check bits for a codeword of the code $C$ and of the CRC, respectively. Then, from (29) of Appendix D (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm), we have

$$t_C(s, n) = 7.5n + f(s),$$

where $f(8) = 33.5$, $f(16) = 51$, $f(32) = 165.5$, and $f(64) = 372$. From Algorithms 2a and 2b, the two check tuples are given by $P_1 = (Z + U_1X^r) \bmod M$ and $P_2 = Y_2 + F(Y_1)$. The first component of $t_C(s, n)$ is $7.5n$ and represents the cost of computing $Z$ and $Y = (Y_1, Y_2)$, while the second component $f(s)$ is the cost of computing $(Z + U_1X^r) \bmod M$ and $Y_2 + F(Y_1)$. The first component varies as a linear function of the tuple count $n$, while the second component $f(s)$ depends only on the tuple size $s$ and not on $n$. Thus, $f(s)$ is a transient component whose contribution becomes negligible for large $n$.

For the CRC, from (30) of Appendix D (which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm), we have

| $s$ \ $ns$ | $2^9$ | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ | $2^{15}$ |
|---|---|---|---|---|---|---|---|
| 8 | 46.2 | 46.6 | 46.8 | 46.9 | 46.9 | 47.0 | 47.0 |
| | 8.02 | 7.76 | 7.63 | 7.57 | 7.53 | 7.52 | 7.51 |
| | 5.76 | 6.00 | 6.13 | 6.20 | 6.23 | 6.25 | 6.26 |
| 16 | 44.0 | 44.8 | 45.1 | 45.3 | 45.4 | 45.5 | 45.5 |
| | 4.55 | 4.15 | 3.95 | 3.85 | 3.80 | 3.77 | 3.76 |
| | 9.69 | 10.8 | 11.4 | 11.8 | 12.0 | 12.0 | 12.1 |
| 32 | 43.3 | 44.0 | 44.4 | 44.6 | 44.7 | 44.7 | 44.7 |
| | 4.46 | 3.17 | 2.52 | 2.20 | 2.04 | 1.96 | 1.92 |
| | 9.72 | 13.9 | 17.6 | 20.3 | 21.9 | 22.9 | 23.4 |
| 64 | 43.0 | 43.7 | 44.0 | 44.2 | 44.3 | 44.3 | 44.4 |
| | 6.75 | 3.84 | 2.39 | 1.66 | 1.30 | 1.12 | 1.03 |
| | 6.37 | 11.4 | 18.4 | 26.6 | 34.0 | 39.6 | 43.1 |

Fig. 5. Operation count per byte of the CRC, operation count per byte of the code $C$, and the ratio of the above two numbers.

$$t_{CRC}(s,n) = 5.5ns + 3n - g(s),$$

where $g(8) = 52$, $g(16) = 93$, $g(32) = g(64) = 90$. For example, let $s = 8$ and $n = 64$, i.e., $ns = 2^9 = 512$ bits. Then, $t_C(8, 64) = (7.5)(64) + 33.5 = 513.5$, i.e., the code $C$ needs 513.5 operations to process 512 bits. Thus, the operation count per byte of the code $C$ is $(8)(513.5)/512 = 8.02$. Similarly, it can be shown that the operation count per byte of the CRC is 46.2. Then, the ratio of the byte operation counts of the CRC and the code $C$ is $46.2/8.02 = 5.76$, i.e., the code $C$ is 5.76 times faster than the CRC. The triplet $(46.2, 8.02, 5.76)$ for the pair $(s, ns) = (8, 2^9)$ is recorded in the left top part of Fig. 5. Triplets for other pairs $(s, ns)$ are similarly obtained.

The results for software complexity of these two codes are summarized in Fig. 5, where $n$ is the total number of $s$-tuples in a codeword, i.e., the total codeword length is $ns$ bits. Here, we consider a wide range of codeword lengths: from $2^9$ to $2^{15}$ bits (i.e., from 64 to 4,096 bytes). Each cell has three numbers: The first number is the operation count per byte of the CRC, the second number is the operation count per byte of the code $C$, the third number is the ratio of the above two numbers and represents the speed improvement of the code $C$ compared to the CRC.

From Fig. 5, as expected, the byte operation count of the CRC slightly decreases when $s$ increases because processing of larger tuples reduces loop overhead. The CRC's operation count also slightly decreases with decreasing $n$ due to the negative term $-g(s)$ in $t_{CRC}(s,n)$. Note that the operation count of the CRC varies only slightly over a wide range of the tuple size $s$ and of the codeword length $ns$. In contrast, the operation count of the code $C$ varies much more as a function of $s$ and $ns$. Further, for each tuple size $s$, the code $C$ is faster for longer codeword length $ns$. This is desirable because speed is more important for longer messages. The reason for the speed variation of the code $C$ is the contribution from the transient term $f(s)$ to the code overall speed. This contribution is noticeable (negligible) if the codewords are short (long). For smaller tuple size $s$ (such as $s = 8$ and 16), the transient term is smaller. Thus, the overall speed variation (as a function of $ns$) of the code $C$ is also smaller. For larger $s$ (such as $s = 32$ and 64), the transient term is greater, resulting in more speed variation (as a function of $ns$) for the code $C$. From Fig. 5, the code $C$ is substantially faster than the CRC, especially for the tuple size $s = 32$ or 64 bits and the code length $ns \geq 2^{13}$ bits $= 1{,}024$ bytes. In particular, if the code length is $ns = 2^{15}$ bits $= 4{,}096$ bytes, then the code $C$ is 23.4 and 43.1 times faster than the CRC when $s$ is 32 and 64 bits, respectively.

| | | | |
|---|---|---|---|
| Code C | 7.5 | 7 | 4 |
| | 0 | 4 | 6144 |
| CRC | 47 | 43 | 6 |
| | 0 | 4 | 512 |

Fig. 6. Operation count per byte and table size in bytes.

**Remark 8.**

1. In Example 4, we derive the operation count expressions $t_C(s,n)$ and $t_{CRC}(s,n)$ for the special case $h = 16$ check bits (when the codes are implemented in $C$ programming language). There, we also assume that the code $C_1$ used in the construction of the code $C$ is the extended Hamming code of length $s$. No such $C_1$ code is needed for the CRC. However, from Figs. 3 and 4, the same expressions also hold true for other values of $h$ and for other codes $C_1$, but with different transient terms that are now denoted as $f(s,h,C_1)$ and $g(s,h)$ to reflect the their dependency on $s$, $h$, and $C_1$. Thus, in general, the software operation counts required to compute the $h$ check bits for a codeword (which consists of $n$ tuples, each tuple is $s$ bits) of these two codes are:
$$t_C(s,n,h,C_1) = 7.5n + f(s,h,C_1)$$
$$t_{CRC}(s,n,h) = 5.5ns + 3n - g(s,h),$$
where the transient terms $f(s,h,C_1)$ and $g(s,h)$ are independent of $n$ and their contributions become negligible when $n$ is large enough. Thus, for large $n$, we have
$$\frac{t_{CRC}(s,n,h)}{t_C(s,n,h,C_1)} \approx \frac{5.5ns + 3n}{7.5n} \approx \frac{5.5ns}{7.5n} = 0.73s,$$
which is an estimate of the speed improvement of the code $C$ compared to the CRC. Again, for large $n$, the code $C$ needs approximately 7.5 operations to process one $s$-tuple or $60/s$ operations per byte. Recall that, in general, the code $C$ is faster than the CRC by the factor $O(s)$. Thus, we have $O(s) \approx 0.73s$ when these error-detection codes are implemented in C programming language.
2. In Fig. 5, we show, without using table lookup, the speed performance of the code $C$ and the CRC, with $h = 16$ check bits. Now, we discuss table-lookup implementations for the same codes. For concreteness, here we assume that each tuple

$Q_i$ has size $s = 8$ bits, as is often used in table-lookup implementations of common CRCs. Larger values of $s$ can be similarly handled, but they result in much larger table sizes. The results are shown in Fig. 6 (whose detailed derivation is given in Appendix D.1, which can be found on the Computer Society Digital Library at http://computer.org/tc/archives.htm). Note that, because $s = 8$ is a small value, the transient terms $f(s, h, C_1)$ and $g(s, h)$ are also small compared to the code overall operation counts. Thus, we estimate the overall operation counts by omitting these transient terms. In particular, the second column shows that, without using table lookup, the code $C$ and the CRC use 7.5 and 47 operations per byte, respectively. The exact values, which vary from 7.51 to 8.02 (for the code $C$) and from 46.2 to 47 (for the CRC), are recorded in Fig. 5. The estimated operation counts and table sizes are shown in Fig. 6. As expected, the operation counts become smaller at the cost of larger tables.

## 5 Summary and Conclusions

We develop Algorithm 1 for generating a large and general family of binary error-detection codes. This algorithm has two key parameters, $s$ and $r$, where $s$ is the size of each tuple and $r$ is the degree of the modulating polynomial $M$. Algorithm 1 is expressed in general and abstract form to facilitate the mathematical development of the resulting code $C$. Error-detection codes used in practice are often systematic. Thus, Algorithm 1 is transformed into systematic versions to yield Algorithm 1a (if $r \leq s$) and Algorithm 1b (if $r > s$).

A variety of error-detection codes (such as CRCs, checksums, and other codes listed in Fig. 1) are developed over the years for applications that require reliable communication or storage. These codes are traditionally considered as unrelated and independent of each other. They also differ considerably in performance and complexity. More complex codes such as CRCs are stronger codes (with minimum distance $d = 4$), whereas simple checksums such as block-parity codes are weaker codes (with $d = 2$). In Section 3, we show that all these diverse codes (from CRCs to checksums), as well as other linear and nonlinear codes, are special cases of Algorithm 1. Thus, these seemingly unrelated codes, which are independently developed over many years, come from a single algorithm.

From Fig. 1, CRCs have the best error-detection capability, but introduce the longest encoding delay. In this paper, we then introduce some non-CRC codes that have good error-detection capabilities as well as fast encoding. In Section 4, we present Algorithm 2a (for $r \leq s$) and Algorithm 2b (for $r > s$), which are fast versions of Algorithm 1a and Algorithm 1b, respectively. These two fast algorithms produce only non-CRC codes. Further, some of these non-CRC codes are not only fast but also reliable. To achieve the minimum distance $= 4$ using $h$ check bits, CRC length can be up to $2^{h-1} - 1$ bits, while the length of some non-CRC codes can be up to $2^{h-1}$ bits (i.e., they are fast versions of perfect codes). We compare the computational complexity of these CRCs and non-CRC codes using methods that require no table lookup. For long messages, the non-CRC codes can be faster than the CRCs by the factor $O(s)$. Further, $O(s) \approx 0.73s$ when these codes are implemented in C programming language. Finally, with the use of table lookup, the operation counts are reduced at the cost of precomputed tables.

## Acknowledgments

This work was supported in part by the US Office of Naval Research.

## References

[1] D. Bertsekas and R. Gallager, *Data Networks,* second ed. Englewood Cliffs, N.J.: Prentice Hall, 1992.
[2] A. Binstock and J. Rex, *Practical Algorithms for Programmers.* Reading, Mass.: Addison-Wesley, 1995.
[3] P. Farkas, "Comments on 'Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes'," *IEEE/ACM Trans. Networking,* vol. 3, no. 2, pp. 222-223, Apr. 1995.
[4] D.C. Feldmeier, "Fast Software Implementation of Error Detection Codes," *IEEE/ACM Trans. Networking,* vol. 3, no. 6, pp. 640-651, Dec. 1995.
[5] J.G. Fletcher, "An Arithmetic Checksum for Serial Transmissions," *IEEE Trans. Comm.,* vol. 30, pp. 247-252, Jan. 1982.
[6] J.G. Fletcher, *ACM Computing Rev.,* vol. 36, no. 1, p. 66, Jan. 1995.
[7] T. Klove and V. Korzhik, *Error Detecting Codes: General Theory and Their Application in Feedback Communication Systems.* Kluwer Academic, 1995.
[8] F.J. MacWilliams and N.J. A. Sloan, *The Theory of Error-Correcting Codes.* New York: North-Holland, 1977.
[9] A.J. McAuley, "Weighted Sum Codes for Error Detection and Their Comparison with Existing Codes," *IEEE/ ACM Trans. Networking,* vol. 2, no. 1, pp. 16-22, Feb. 1994.
[10] G.D. Nguyen, "A General Class of Error-Detection Codes," *Proc. 32nd Conf. Information Sciences and Systems,* pp. 451-453, Mar. 1998.
[11] A. Perez, "Byte-Wise CRC Calculations," *IEEE Micro,* vol. 3, pp. 40-50, June 1983.
[12] T.V. Ramabadran and S.S. Gaitonde, "A Tutorial on CRC Computations," *IEEE Micro,* vol. 8, pp. 62-75, Aug. 1988.
[13] D.V. Sarwate, "Computation of Cyclic Redundancy Checks via Table-Lookup," *Comm. ACM,* vol. 31, no. 8, pp. 1008-1013, Aug. 1988.
[14] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of Checksums and CRC's over Real Data," *IEEE/ACM Trans. Networking,* vol. 6, no. 5, pp. 529-543, Oct. 1998.
[15] J.L. Vasilev, "On Nongroup Close-Packed Codes (in Russian)," *Problemi Cybernetica,* vol. 8, pp. 337-339, 1962.

**Gam D. Nguyen** received the PhD in electrical engineering from the University of Maryland, College Park, in 1990. He has been at the US Naval Research Laboratory, Washington, DC, since 1991. His research interests include communication systems and networks.